User's guide for `ranlxs` and `ranlxd` v3.4

Martin Lüscher May 2019

The C programs described in this guide provide a highly efficient implementation of the RANLUX random number generator [1,2]. There are two interfaces to the basic algorithm, `ranlxs` and `ranlxd`, delivering single- and double-precision random numbers. Details on the underlying algorithms and their implementation can be found in the accompanying notes [3].

Machine requirements

For the programs to work as intended, the computer must be able to handle single- and double-precision floating-point numbers according to the IEEE 754 standard. Moreover, it is taken for granted that the C data type `int` has size greater or equal to 4 and that one of the integer data types `int`, `long` or `long long` has size greater than 6.

Current processors, operating systems and compilers usually satisfy all these conditions. The programs however check whether this is indeed the case and terminate with an informative error message if not.

Files

The program files included in the package are:

```
ranlux.h
ranlxs.c
ranlxd.c
ranlux_common.c
testlx.c
timelx.c
```

The last two are main programs that allow the correct functioning and performance of the `ranlxs` and `ranlxd` generators to be checked. It is possible to use the two generators individually or concurrently. The programs contained in `ranlux_common.c` are called by the functions in `ranlxs.c` and `ranlxd.c`, but are not intended to be used otherwise.

The externally accessible functions defined in the program files `ranlxs.c` and `ranlxd.c` are

`ranlxs`	`ranlxd`
`rlxs_init`	`rlxd_init`
`rlxs_size`	`rlxd_size`
`rlxs_get`	`rlxd_get`
`rlxs_reset`	`rlxd_reset`

`ranlxs` and `ranlxd` are the main subroutines, all other functions being utility programs that provide access to the state of the generators. For illustration a GNU-style Makefile that compiles all programs is included in the package.

Compilation

The programs are written in C and comply with the C99 standard of the language. No special options are required and a command like

```
cc [options] ranlxs.c ranlxd.c ranlux_common.c testlx.c -o testlx
```

should compile the programs and produce the executable `testlx`. The programs are expected to work correctly, even with the most aggressive optimization options, but it is recommended to check this by running `testlx`. This program performs a number of tests and reports the results to `stdout`.

Initialization

Both generators must be initialized by calling the functions `rlx*_init` (with `*` being equal to `s` or `d` as appropriate). The synopsis is

```
#include "ranlux.h"
void rlx*_init(int level,int seed);
```

where the seed is an arbitrary integer in the range from 1 to $2^{31}-1$. Different seeds are guaranteed to result in different sequences of random numbers. The initialization programs set the generators to a definite state and the generated sequences of random numbers are thus reproducible.

The luxury level has to be equal to $0, 1$ or 2, with only the two higher values being permitted in the case of `ranlxd`. This parameter controls the statistical quality of the random numbers generated. For many applications, including large scale Monte Carlo simulations, the lowest level should already be adequate. Increasing the level by 1 reduces the residual correlations by several orders of magnitude [1,3].

Random number generation

The functions `ranlxs` and `ranlxd` generate random numbers of type `float` and `double` respectively. The synopsis are

```
#include "ranlux.h"
void ranlxs(float *r,int n);
```

and

```
#include "ranlux.h"
void ranlxd(double *r,int n);
```

Both functions generate `n` new random numbers and assign them to the first `n` elements of the array `r`. It is left to the user to ensure that `r` is declared appropriately, i.e. no check on the array bounds is made. A typical code then looks like

```
#include "ranlux.h"
#define LVEC 12
float rvec[LVEC];
.
.
rlxs_init(1,3456);
ranlxs(rvec,LVEC);
```

The random numbers generated by `ranlxs` are uniformly distributed in the range

$$x/2^{24}, \qquad x = 0, 1, 2, \ldots, 2^{24}-1.$$

All numbers in this range are exactly representable on computers that pass the tests performed by the initialization program.

In the case of `ranlxd`, the generated random numbers are uniformly distributed in the extended range

$$x/2^{48}, \qquad x = 0, 1, 2, \ldots, 2^{48} - 1.$$

They are also exactly representable, but an important detail to keep in mind is the fact that the mantissa of IEEE 754 double-precision floating-point numbers have 53 bits, i.e. the 5 least significant bits of the generated numbers are always equal to zero on machines complying with the standard.

I/O routines

While the states of the generators are not directly accessible, it is possible to extract the complete information on the current states through the functions `rlx*_get` (where the `*` again stands for `s` or `d` as appropriate). The synopsis is

```
#include "ranlux.h"
void rlx*_get(int *state);
```

On output the array `state` contains the desired information in an encoded form. The array must be declared to have `n=rlx*_size()` or more elements.

At a later stage in the calling program or in another program, the generators may then be reset to the state defined by the array `state` by invoking the functions

```
#include "ranlux.h"
void rlx*_reset(int *state);
```

These programs check that the data passed to them are sane and exit with an error message if not.

Table 1. Execution times in nanoseconds per random number measured on a machine with Intel Core i3-7100U (2.4 GHz) processor

program	code	level 0	level 1	level 2
`ranlxs`	C99	8.6	12.8	21.5
`ranlxs`	SSE2	6.2	8.9	14.5
`ranlxs`	AVX2	4.8	6.0	8.3
`ranlxd`	C99	–	21.4	38.9
`ranlxd`	SSE2	–	14.5	25.6
`ranlxd`	AVX2	–	8.2	12.9

Using x86-64 SIMD instructions

The functions that update the states of the generators include SIMD inline assembly code that may optionally be used on x86-64 machines. To enable the assembly code, one of the macros `SSE2` or `AVX2` must be defined at compilation time. With the GNU C compiler, this can be achieved by setting the option `-DSSE2` or `-DAVX2` as in

```
gcc -O2 -DSSE2 ranlxs.c ranlxd.c ranlux_common.c timelx.c -o timelx
```

for example. The Intel compiler `icc` understands the inline assembly code too and can be used in place of `gcc`. If both `SSE2` and `AVX2` are defined, the former has no effect.

Most x86-64 machines support SSE2 instructions nowadays, but the full AVX2 instruction set is only supported by the more recent Intel and AMD processors at this time. The correct functioning of the compiled binaries should in any case be checked by running the program `testlx`.

Timing

The computer time required to generate new random numbers depends on the machine, the luxury level and the compiler options. Some benchmark results, obtained with the GNU C compiler and the optimization option `-O2`, are reported in table 1.

References

[1] M. Lüscher, A portable high-quality random number generator for lattice field theory simulations, Comp. Phys. Comm. 79 (1994) 100

[2] F. James, RANLUX: a Fortran implementation of the high-quality pseudo-random number generator of Lüscher, Comp. Phys. Comm. 79 (1994) 111 [E: *ibid.* 97 (1996) 357]

[3] M. Lüscher, Algorithms used in `ranlxs` and `ranlxd` v3.4 (May 2019)